Ginger: Control Independence Using Tag Rewriting

Andrew Hilton and Amir Roth
Department of Computer and Information Science, University of Pennsylvania
{adhilton,amir}@cis.upenn.edu

ABSTRACT

The negative performance impact of branch mis-predictions can be reduced by exploiting control independence (CI). When a branch mis-predicts, the wrong-path instructions up to the point where control converges with the correct path are selectively squashed and replaced with correct-path instructions. Instructions beyond the convergence-point—the branch's control-independent (CI) instructions—are spared from squashing. Exploiting CI requires updating the input data dependences of CI instructions to reflect the selective removal and insertion of logically older instructions and transitively re-dispatching those CI instructions whose inputs have changed. This capability is generally called out-of-order renaming. Previously proposed CI designs use out-of-order renaming schemes that either consume excessive rename/dispatch bandwidth, can only be applied in limited cases, or incur a cost even when the branch would be correctly predicted.

Ginger is a CI design that is both general and bandwidth efficient. Ginger implements out-of-order renaming using tag rewriting, re-linking the input dependences of CI instructions as they sit in the window. To do this, Ginger halts the pipeline uses the idle map table read and write ports and the issue queue match lines and write lines to perform a register-tag "search-and-replace" operation. After a few cycles, the pipeline restarts and execution resumes with correct data dependences. Cycle-level simulation shows that Ginger out-performs previous CI designs, yielding geometric mean speedups over an aggressive non-CI processor of 5%, 12%, and 11%—on SPECint2000, MediaBench, and CommBench—with speedups of 15% or greater on 11 of 46 programs.

Categories and Subject Descriptors

C.1.3. [**Processor Architectures**]: pipelined processors.

General Terms

Design, performance, measurement.

Keywords

Branch mis-predictions, control independence, out-of-order renaming, selective re-dispatch.

1. INTRODUCTION

Branch mis-predictions limit the performance of conventional superscalar processors. Because these processors support only in-order renaming, mis-prediction recovery requires squashing, re-fetching, and re-renaming all instructions younger than the branch. Even processors with moderate depth pipelines like IBM's POWER5 [26] squash up to 50 instructions per mis-prediction, and often many more. The result is substantial loss in fetch and rename throughput, which in turn limits commit throughput.

The cost of full-squash mis-prediction recovery is even higher when one observes that control flow following many mis-predicted branches quickly converges with the correct path. A significant amount of waste could be avoided if squashing could be selectively applied only to instructions up to this convergence point—those that actually correspond to the wrong path—sparing the post-convergence instructions. Designs that target this effect are said to exploit control independence (CI) [22]. The post convergence instructions are called control independent (CI) instructions. In the general case, exploiting CI also requires inserting pre-convergence correct-path instructions before the spared CI instructions.

The primary CI implementation challenge is constructing and enforcing correct input data dependences for the CI instructions that selective squashing spares. Because (register) dependences are constructed by renaming and because these instructions are physically renamed before instructions that are logically older, this capability is called out-of-order renaming. Proposed CI designs distinguish themselves by their out-of-order renaming implementations. An early proposal [22]—that here we call Walker—walks the CI instructions, re-renames them and transitively re-dispatches those whose inputs have changed from a recovery buffer. Because the number of CI instructions is typically large, Walker consumes significant rename bandwidth. Selective Branch Recovery (SBR) [8] does not physically re-rename CI instructions but achieves the same effect by converting the wrong-path instructions into moves and re-dispatching them and their dependent CI instructions, again from a recovery buffer. SBR consumes less rename bandwidth than Walker but can exploit CI only in cases that do not require splicing instructions into the window.

Walker and SBR both exploit CI reactively, explicitly doing work only after an actual branch mis-prediction is detected. A third design called Skipper [4] exploits CI proactively. When Skipper identifies a low-confidence branch, it defers speculative fetch. Until the branch executes, Skipper fetches and executes CI instructions from its predicted convergence point, "holding" those that may depend on the still-to-be fetched correct-path instructions using register place-holders. When the branch resolves, the correct-path instructions are fetched into pre-allocated window space and executed and the dependence place-holders are released. Skipper avoids re-rename and re-dispatch but delays some CI instructions even when the branch would have been correctly predicted—the common case even for low confidence branches.

This paper proposes Ginger, a microarchitecture that performs reactive CI in a general and bandwidth efficient way. Ginger combines aspects of these previous schemes with a different implementation of out-of-order renaming: tag rewriting [31]. Like Skipper, Ginger proactively identifies low-confidence branches and reserves window space for control-dependent instructions. But Ginger does this only to simplify resource management, it fetches

ISCA07, June 9–13, 2007, San Diego, California, USA.

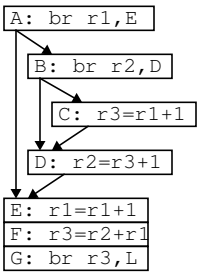
and executes speculatively and does not delay any CI instructions. Like Walker, Ginger reacts to mis-predictions by physically squashing the wrong-path instructions and replacing them with correct-path ones. And like Walker and SBR, Ginger re-dispatches CI instructions whose inputs have changed from a recovery buffer. However, Ginger performs out-of-order renaming differently. After selective instruction removal and insertion, Ginger temporarily halts the pipeline. Co-opting the map table read ports normally used for renaming, the issue queue write ports normally used for dispatch, and the issue queue associative match ports normally used for scheduler wakeup, Ginger walks (a subset of) the logical registers and rewrites the input dependences of instructions as they sit in the issue queue and the recovery buffer. After this procedure—which usually takes a few cycles—processing resumes with all data dependences linked correctly. Ginger supports reactive CI in general cases, does not walk all CI instructions to do so, and leverages existing hardware.

Beyond out-of-order renaming, supporting CI potentially requires invasive changes to other aspects of the pipeline. Insertion and removal of instructions into and from the middle of the window may require modified resource allocation schemes. CI may also require new store-load forwarding and memory-disambiguation mechanisms, as conventional mechanisms rely on in-order load-store queues. CI also interferes with conventional branch predictors, which assume complete and correct histories of previous branch outcomes. CI can even introduce false branch mis-predictions. Nested or otherwise simultaneously active instances of CI further complicate each of these aspects. Each previously proposed CI design explicitly addresses some subset of these challenges. Ginger uses conventional resource allocation, adapts recently proposed load-store queue designs for CI-aware store-load forwarding and memory disambiguation, and adds support for CI-aware branch prediction and false mis-prediction suppression.

Experiments using the SPECint2000, MediaBench and CommBench benchmark suites show that Ginger exploits CI more effectively than re-implementations of these previously proposed schemes. On a 4-way issue processor with a 512-entry re-order buffer and a large branch predictor, Ginger produces average speedups of 5%, 12%, and 11% over a conventional non-CI baseline; with speedups higher than 15% on 11 of 46 programs.

The main contributions of this paper are:

- An out-of-order renaming design using tag rewriting.
- Ginger, a microarchitecture that uses this low-overhead out-of-order renaming scheme to exploit CI.
- An evaluation that compares Ginger to three previously-proposed CI schemes.



A's **convergence PC** is E

A's **convergence distance** is 3 instructions

A's **register output set** is (r2, r3)

B,C & D are **control-dependent (CD)** on A

E,F & G are **control-independent (CI)** of A

E is **data-independent (CI_{DI})**

F & G are **data-dependent (CI_{DD})**

Figure 1. CI code example and terminology.

2. CONTROL INDEPENDENCE (CI)

This section uses an example to illustrate the differences between four CI designs: Walker [22], SBR [8], Skipper [4] and Ginger. Figure 1 shows a code sequence comprising 7 instructions (A–G) and 3 branches (A, B and G). Branch A is low-confidence and is the one to which CI is applied. Instructions B–D are control-dependent (CD) on branch A; E–G are control independent (CI) of it. Branch A's convergence PC is E, its convergence distance is three instructions, and its register output set—the registers that may be written by its CD instructions along any path—is r2 and r3. We assume the processor is given or can predict these three quantities [4, 7]. A's CI instructions can be further sub-categorized. E is data-independent of A (CI_{DI})—it does not read (either directly or transitively) any register written by A's CD instructions. F and G are data-dependent (CI_{DD})—F reads r2 which is written by CD instruction C, and G reads r3 which is written by F. The running example assumes that A is mis-predicted not-taken and that B-D are wrong-path CD instructions.

2.1. Walker

Walker is a CI design that repairs and re-dispatches CI_{DD} instructions by re-renaming all CI instructions.

Figure 2A shows the state of the processor when branch A's mis-prediction is detected. Instructions A–G are in the window. In addition to the active map table, the processor has a recovery checkpoint for A. In Figure 2B, Walker restores A's recovery checkpoint and selectively squashes wrong-path CD instructions B–D by resetting the dispatch pointer. Walker supports both removal and insertion of instructions from and into the middle of the window. If A instead were mis-predicted taken, B–D would be spliced into the window and renamed using the restored map-table.

Removing C changes the definition of r2 with respect to instruction F and so F's dependence on r2 must be repaired. If F has already issued it must re-issue with this repaired input, as must G which depends on F. In Figure 2C, Walker walks A's CI instruc-

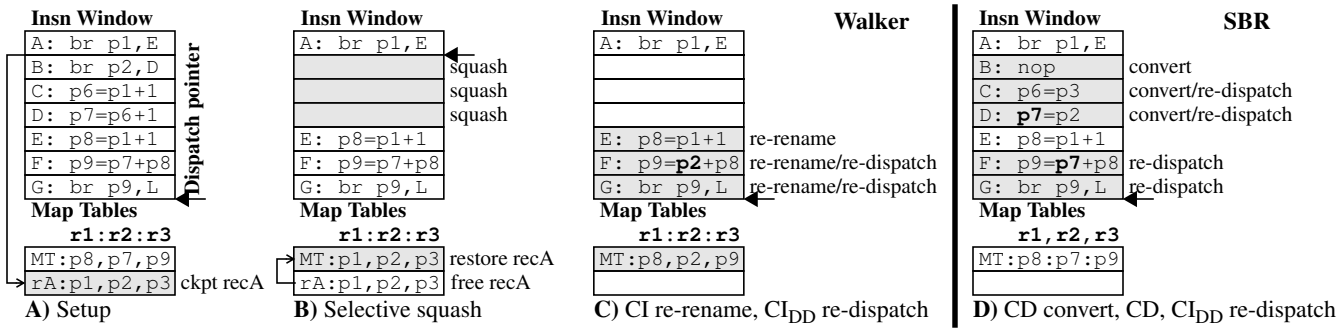


Figure 2. Walker and SBR action diagrams.

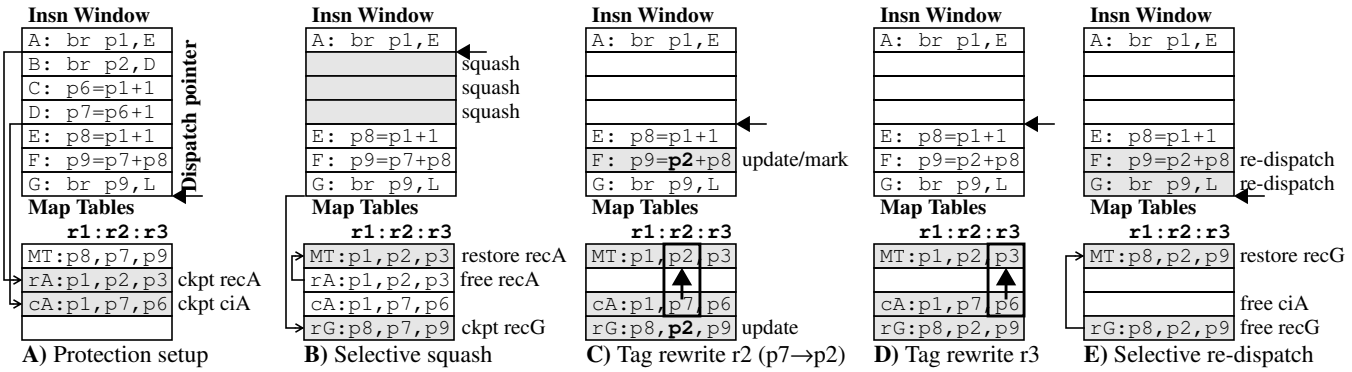


Figure 3. Ginger Action Diagrams.

tions E–G, re-renames them, and selectively and transitively re-dispatches F and G. Re-renaming differs from (initial) renaming in that it only fixes input dependences, it does not re-assign outputs.

Pros and cons. Walker is reactive—it does not slow down the common case of the branch being correctly predicted—and can exploit CI in general cases. Its disadvantage is that it consumes bandwidth re-renaming CI_{DI} instructions.

2.2. Selective Branch Recovery (SBR)

SBR is a reactive CI scheme that supports only selective instruction removal—it could not exploit CI if A were mis-predicted taken and B–D had to be inserted. Removal-only CI—also called exact convergence—occurs naturally in 'if-then' (no 'else') conditionals and some loops. It can also be induced in some cases.

Figure 2D shows SBR's actions. When branch A mis-predicts, SBR repairs CI_{DD} instruction input dependences not by re-linking their physical register tags, but rather by converting register-writing wrong-path CD instructions into special moves and re-dispatching them. SBR converts C into a move that copies the value in the physical register mapped to r3 *prior to* C (p3) to the register mapped to r3 *by* C (p6). D becomes a move from p2 to p7. C, D, and CI_{DD} instructions F and G re-dispatch from a recovery buffer.

Pros and cons. SBR is simple. Its drawbacks are its limited applicability and the fact that it consumes bandwidth re-dispatching and re-issuing wrong-path CD instructions as moves. These moves also add dataflow latency.

2.3. Skipper

Skipper is a proactive CI microarchitecture. Unlike Walker, SBR and Ginger—which repair mis-predictions after the fact—Skipper side-steps mis-predictions. Skipper identifies a low-confidence branch and defers fetching its CD instructions until its outcome is known. In the meantime, it fetches and executes its CI instructions. When the branch executes, the correct-path CD instructions dispatch into pre-allocated window space. To avoid dependence repair and re-issue, Skipper pre-synchronizes CI_{DD} instructions by allocating physical register place-holders to the branch's output-set registers. After the correct-path CD instructions dispatch, Skipper dispatches additional moves that write the place-holder registers and release the waiting CI_{DD} instructions.

Pros and cons. Skipper does not need re-rename/re-dispatch machinery and does not waste bandwidth on wrong-path CD instructions. However, it delays CI_{DD} instructions until the branch executes even when that branch is correctly predicted. In doing so, it also uses issue queue space inefficiently. These overheads make Skipper profitable only on branches that mis-predict frequently.

2.4. Ginger

Ginger—the architecture we propose here—combines elements of these previously proposed techniques. Like Walker, it supports general reactive CI—it can both selectively remove and insert CD instructions—and does so by physically re-linking the input dependences of CI_{DD} instructions. Like Walker and SBR, Ginger selectively re-dispatches CI_{DD} instructions from a recovery buffer. And like Skipper, it takes proactive steps when dispatching a low-confidence branch to simplify future selective recovery.

Figure 3 shows the actions taken by Ginger. When Ginger dispatches branch A, it predicts its convergence PC (E) and distance (3 instructions). Ginger continues fetching and executing speculatively, but actively looks for E. When E renames, if A has not yet resolved and if there is enough space in the window for A's alternate path instructions—here, because A is predicted not-taken additional window space is not needed—then Ginger sets up potential selective-squash recovery for A by taking a second map-table checkpoint. In Figure 3A, branch A has two checkpoints—the conventional recovery checkpoint (recA) that was taken after it was renamed, and the CI checkpoint (ciA) that was taken before its convergence PC (E) was renamed.

When A mis-predicts (Figure 3B), Ginger checkpoints the current map table (recG) so that it can return to it later. It also restores A's recovery checkpoint (recA) and resets the dispatch pointer to A, effectively squashing wrong-path CD instructions B–D. If A were instead mis-predicted taken, Ginger would dispatch correct-path CD instructions B–D into the pre-allocated gap.

When CD instruction removal and insertion completes, Ginger temporarily halts the pipeline and walks A's register output set (r2 and r3). For each register, Ginger reads the old mapping from A's CI checkpoint and the new mapping from the active map table and performs a search-and-replace on the input tags of every instruction in the issue queue and recovery buffer. The issue queue's existing wakeup match lines and dispatch write lines are co-opted to perform this function. In parallel, Ginger also updates all logically younger map-table checkpoints. In Figure 3C, tag rewriting for r2 replaces all instances of p7 with p2. r2's "successful" tag rewrite for instruction F flags F for re-dispatch. Checkpoint recG's r2 entry is also re-written. In Figure 3D, tag rewriting for r3 replaces instances of p6 with p3, but no instructions or map-table checkpoints are affected.

In Figure 3E, tag re-writing completes. Ginger restores the tail map table checkpoint (recG), and frees A's recovery and CI checkpoints (recA and ciA). Ginger also re-activates the front-end pipeline and the scheduler, which restarts with all dependences re-linked correctly. F and G then re-dispatch from a recovery buffer.

Pros and cons. Ginger can apply reactive CI in general cases and consumes re-rename/re-dispatch bandwidth in proportion to the sum of the branch's CI_{DD} instructions and output set registers rather than to its CI instructions, which are typically much more numerous. However, tag rewriting consumes issue bandwidth, and Ginger uses more map-table checkpoints than the other schemes.

3. GINGER MICROARCHITECTURE

This section fleshes out the Ginger design which was sketched in Section 2.4. Section 3.1 describes Ginger's large instruction window support. Sections 3.2–3.7 describe Ginger's core mechanisms including tag rewriting and selective re-dispatch. Sections 3.8–3.9 describe Ginger's interactions with branch prediction.

3.1. Large Instruction Window

Ginger improves the utilization of a large instruction window and also needs a large window to operate effectively. The critical structures of a large window are the register file, load-store queue (LSQ), issue queue, and register map table checkpoints—the reorder buffer (ROB) itself is not critical. Register file latency can be controlled using replication and hidden by deeper bypass networks; Ginger uses scalable designs for the LSQ, issue queue.

To simplify resource management, Ginger assumes that there are the same numbers of ROB entries, LSQ entries, and renaming registers. Ginger allocates and de-allocates ROB, LSQ, and physical register file entries together—potentially in large chunks—by adding to or subtracting from a single pointer which corresponds to a position in the ROB, LSQ, and register free list.

Load-store queue. Scaling a traditional associative LSQ is difficult. Associative search latency both scales poorly and shows up as load execution latency due to store-load forwarding. Ginger uses a non-associative LSQ which scales more easily and meshes with its tag-rewriting implementation of out-of-order renaming.

For store-load forwarding, Ginger uses store queue index prediction (SQIP) [25]. At decode, a forwarding predictor maps each load to the PC of the most likely forwarding store. A rename stage store map table maps each store PC to the store queue index of its most recent dynamic instance. The store map table is analogous to the register map table, and is checkpointed and recovered in parallel. The scheduler uses the store queue index as a tag to synchronize the load with the store. When the load executes, it accesses the store queue directly at this predicted index and forwards from the store on an address match. To verify forwarding speculation, loads re-execute in order prior to commit. Mis-speculation is detected when the re-executed value does not match the value obtained during out-of-order execution [3]. To conserve data cache bandwidth, the store vulnerability window (SVW) mechanism filters over 99% of would-be re-executions [24].

Issue queue. Ginger improves the utilization of a large window and also uses selective re-issue; both factors prefer a larger issue queue. Physically scaling the issue queue is difficult because performance is sensitive to wakeup/select loop latency. Ginger avoids increasing issue queue latency using a hierarchical scheme that combines a small conventional issue queue with a ROB-sized re-dispatch queue. Section 3.6 describes this organization in detail.

Map-table checkpoints. Processors use map-table checkpoints to support fast mis-prediction recovery. Checkpoints are more important in CI processors than they are in conventional processors. A conventional processor serializes mis-predictions, reducing the marginal utility of additional checkpoints. A CI processor parallelizes mis-predictions, preserving and even enhancing marginal checkpoint utility. In Ginger, checkpoints are not only more important than they are in a conventional processor, they are also more expensive. First, tag rewriting requires updating the checkpoints which precludes shadow bit-cell implementations. Then, to apply selective-squash recovery to a given branch Ginger needs two checkpoints, not one. Finally, SQIP requires store map-table checkpoints in addition to register map-table checkpoints. Ginger uses branch confidence to manage a limited number of checkpoints efficiently [1].

3.2. Setup and Branch Protection

To exploit CI for—i.e., to apply selective squash recovery to—a given branch, Ginger must explicitly protect the branch by speculatively performing certain setup tasks when the branch itself and the instruction at its convergence PC are initially renamed.

When Ginger renames a low-confidence branch, it allocates a map-table checkpoint to it to enable fast full-squash recovery. This checkpoint, which even conventional processors take, is the *recovery checkpoint*. Ginger also predicts the branch's convergence PC and distance and waits for the instruction at the convergence PC to arrive at rename. At that time, if the branch has not executed yet and if there is enough space in the window for the branch's alternate-path CD instructions Ginger reserves window space and takes a second map-table checkpoint. This Ginger-specific checkpoint is the *CI checkpoint*. Map-table checkpointing and window-space reservation are one-cycle operations that do not consume instruction processing bandwidth.

If there is no space in the window for the alternate-path CD instructions, or if Ginger renames more instructions than the predicted convergence distance, protection setup fails and selective squash recovery is not applied to this branch.

Protecting nested branches. To simplify protection setup and resource allocation, Ginger protects nested low-confidence branches in a restricted way. In Figure 1, branch B is nested within branch A's CD region. After Ginger renames A, it waits for A's predicted convergence PC (E) to rename. As long as it is actively looking for E, Ginger does not try to independently protect B. If A's setup succeeds, then A's CI checkpoint will also protect B—although if B mis-predicts Ginger selectively squashes A's CD instructions (C and D), not B's CD instruction (C). If A setup fails, neither A nor B is protected.

3.3. Selective Squash Recovery

When a protected branch mis-predicts, Ginger redirects fetch to the branch's actual target PC. While the correct-path CD instructions are fetched, Ginger drains the front-end pipeline—which is full of younger CI instructions—by renaming and dispatching the instructions into the tail of the window. A conventional processor would discard these instructions; Ginger preserves them. When the protected branch's correct-path CD instructions are ready for rename and dispatch, Ginger checkpoints the cur-

rently active map-table so that it can return to it after selective recovery. It then removes the wrong-path CD instructions, restores the branch's map-table recovery checkpoint, and resets the ROB, LSQ, and register free list pointers to the slot immediately following the branch. The correct-path CD instructions are renamed using the restored recovery map-table and dispatch to the window.

3.4. Tag Rewriting

After removing the wrong-path CD instructions and inserting the correct-path CD instructions, Ginger repairs the input data dependences of the younger CI_{DD} instructions. More accurately, it attempts to repair the input data dependences of *all* younger CI instructions and recognizes those whose inputs actually change as CI_{DD} instructions. Ginger also updates any logically younger map tables that may be used to rename future instructions.

At the end of selective-squash recovery, Ginger has two map tables that correspond to the beginning of the protected branch's CI region. The CI checkpoint, taken before the branch's convergence PC was initially renamed, contains the "old" mappings. The recovery checkpoint, taken after the branch itself was initially renamed and subsequently restored and updated with the mappings of the correct-path CD instructions, contains the "new" mappings. Logically, tag rewriting walks these two map tables in parallel and for every register for which the mappings differ performs an old-to-new-mapping "search-and-replace" operation. The walk over all logical registers can be streamlined to a walk over registers whose mappings differ by tracking the destination registers of actual wrong-path and correct-path CD instructions using a bitvector.

To perform tag rewriting, Ginger uses existing structures and ports that are otherwise used for rename, dispatch, and wakeup. Logically, it is easy to think of Ginger as completely halting the pipeline while it rewrites tags and we will continue to present it this way for now. However, Ginger can also rewrite tags at less-than-full pipeline bandwidth and perform conventional processing with the remaining bandwidth. Section 3.6 discusses this trade-off.

Issue queue tag rewriting. A conventional issue queue already includes tag search-and-replace capabilities. The scheduler uses tag match ("search") lines for wakeup. Dispatch uses tag write ("replace") lines to write new instructions into the issue queue. Ginger implements tag rewriting in the issue queue by combining these two functions. Ginger broadcasts the old-mapping tag on the wakeup match lines. On a match, the corresponding tag and its ready bit are overwritten by the new-mapping tag and its ready bit currently on the write lines. The ready bit for the new-mapping tag is retrieved by reading the physical register ready bitvector.

To support tag rewriting, issue queue tags are extended to include logical—in addition to physical—register names. This is done to disambiguate the situation in which two logical registers essentially swap physical register mappings. Suppose r2's old/new mappings are p7/p9 while r3's are the reverse—because Ginger associates physical registers with ROB slots this scenario can arise naturally. If tag rewriting used only physical register names, then Ginger would first replace all instances of p7 with p9, and then p9 instances with p7. This would not achieve the desired effect. With extended tags, Ginger first replaces instances of r2:p7 with r2:p9 and then r3:p9 instances with r3:p7.

Map-table tag rewriting. To allow new instructions to be renamed correctly, tag rewriting updates all map tables that are logically younger than the branch's convergence point. The active map-table is the branch's updated recovery map-table—it *is* at the branch's convergence point—and so the map-tables that must be updated are all technically checkpoints.

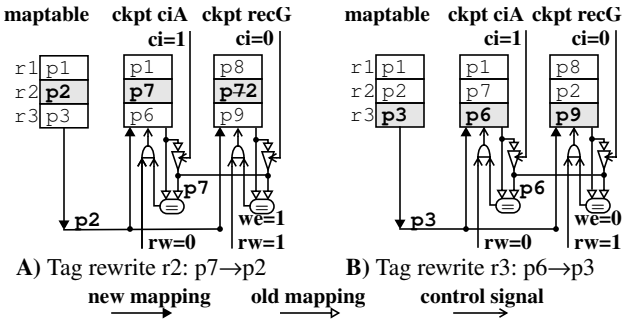


Figure 4. Ginger map-table tag re-writing.

Re-writing a given logical register requires reading and writing the entry for that logical register in all checkpoints, and so the checkpoints are organized as a single, wide structure with multiple banks. In addition to creation/restoration ports, the wide checkpoint contains one read port and one write port for every register tag rewrite performed per cycle. Each read port also feeds a shared bus that all banks observe. Each bank is associated with two signals. The *CI* signal is high for the bank corresponding to the old-mapping CI checkpoint. The *RW* signal is high for all banks corresponding to checkpoints younger than the active map table. Ginger computes these signals before every tag rewriting phase. For each re-written logical register, the CI bank and all RW banks read the corresponding physical register tag. The CI bank's tag is broadcast on the shared bus. Each RW bank locally compares its tag with the CI tag. If the tags match, then the physical register tag from the active map table is written into the checkpoint bank.

Figures 4 shows working examples of map-table tag rewriting. The mappings are those from Figures 3C and 3D. Each diagram shows the active map-table and two checkpoints: A's CI checkpoint (ciA) and the ROB tail checkpoint (recG). Checkpoint recG is younger than the active map-table and is the checkpoint being re-written—its bank's RW signal is 1. In Figure 4A, Ginger tag rewrites r2, replacing instances of p7 (checkpoint ciA's mapping) with p2 (active map-table's mapping). Checkpoint recG is write-enabled because its own mapping for r2 matches p7. In 4B, Ginger rewrites r3, replacing p6 with p3. This time, checkpoint recG is not write-enabled because its own mapping for r3 is p9, not p6.

Re-order buffer tag rewriting. Instructions store the physical register tags overwritten by their destination logical registers in their ROB entries to support both serial mis-prediction recovery and physical register reclamation at commit. As a result, physical register tags must be rewritten in the ROB as well. To do this, Ginger adds associative match/replace ports to the ROB. Because tag rewriting is not latency-critical, the ROB can be segmented with these ports pipelined to traverse one segment per cycle [9].

Freeing map-table checkpoints. When tag rewriting completes, Ginger frees the branch's CI checkpoint. Ginger also frees a branch's recovery and CI checkpoints if the branch executes and resolves correctly. Nested Ginger-protected branches essentially share a single CI checkpoint, and so this checkpoint is freed only when the last in a group of nested protected branches executes.

3.5. Selective Re-Dispatch

Tag rewriting ensures that CI_{DD} instructions that have not yet issued will issue with the correct register inputs. But what about CI_{DD} instructions that have already issued and left the issue queue? How are these notified of tag changes and re-dispatched?

Ginger implements this functionality using a hierarchical approach. A conventional issue queue, which is small enough to support single-cycle wakeup/select, issues instructions to the functional units. A secondary *re-dispatch queue* that contains all in-flight instructions in program order re-dispatches CI_{DD} instructions to the issue queue. Ginger's re-dispatch queue resembles SBR's recovery queue and other previously proposed designs [1, 8, 17, 29]. It differs from these in that it supports associative matching which it uses for both tag rewriting and re-dispatch pseudo-scheduling. It also uses a segmented pipelined design [9].

Re-dispatch mechanics. Instruction re-dispatch uses a pseudo-scheduler that is based on propagated tag changes rather than availability of physical register operands. This functionality means re-dispatch queue entries are slightly different than their issue queue counterparts—they do not have per-operand ready bits, instead they have per instruction re-dispatch bits. A set re-dispatch bit means the instruction should be selected for re-dispatch.

Instructions initially dispatch to the issue queue. As they issue, they are written into the re-dispatch queue at slots corresponding to their ROB positions. The re-dispatch queue monitors tag changes and applies them in the same way as the issue queue. A successful tag change sets the instruction's re-dispatch bit, marking it for future re-dispatch. Every cycle, the re-dispatch scheduler selects instructions whose re-dispatch bits are set and re-dispatches them. It broadcasts the destination tags of these instructions to the re-dispatch queue, waking up dependent instructions and setting their re-dispatch bits.

Re-dispatching instructions clear the physical register ready bitvector entries corresponding to their destinations. However, they do not "un-wakeup" dependent instructions that may still reside in the issue queue because doing so would consume wakeup bandwidth. As a result, there is a gap of vulnerability that manifests when an instruction X re-issues after a dependent instruction Y issues for the first time. Y issues with the wrong input value (from X's initial execution) and when it enters the re-dispatch queue, it is not marked for re-dispatch because X's re-dispatch and wakeup has already taken place. To close this hole, Ginger maintains a per physical register re-dispatch bitvector. A set entry in this bitvector indicates that the instruction writing the physical register has re-dispatched but has not yet re-issued. Re-dispatch sets entries in this bitvector, re-issue clears them. As instructions enter the re-dispatch queue, they read the re-dispatch bitvector entries corresponding to their input operands. If a re-dispatch bit for *any* input is set, the instruction enters the re-dispatch queue with its re-dispatch bit pre-set. The re-dispatch bitvector is analogous to the ready bitvector used by the primary scheduler to track instructions that are ready to issue upon dispatch.

Figure 5 shows the structures and paths Ginger uses to implement selective re-dispatch. The figure shows a scalar rename, dispatch, and issue pipeline for a 2-input 1-output instruction. Re-dispatch contends with primary dispatch. Ginger awards the dispatch slot based on the relationship of the active dispatch pointer to the ROB position of the re-dispatching instruction—oldest wins. In

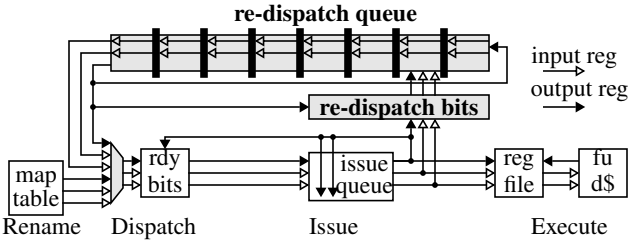


Figure 5. Ginger selective re-dispatch.

most ways, a re-dispatching instruction is indistinguishable from a dispatching instruction. It reads the ready bits corresponding to its inputs, clears the ready bit corresponding to its output, and is written into the issue queue. However, it also sets the re-dispatch bit corresponding to its output and broadcasts its output tag to the re-dispatch queue. On issue, it broadcasts its output tag to the issue queue, sets the ready bit corresponding to its output and also clears the re-dispatch bit corresponding to its output.

Re-dispatch queue design. Because it contains as many slots as the ROB and because the CI_{DD} instructions it affects are not performance-critical, the re-dispatch queue is implemented less aggressively than the issue queue. Ginger's default 512-instruction re-dispatch queue is divided into eight 64-instruction segments and tag broadcast is pipelined over these segments starting at the head [9]. Waking tag-rewriting an instruction in the oldest segment takes 1 cycle, waking an instruction in the youngest segment takes 8 cycles. Instructions that young can typically tolerate this delay.

Managing issue queue slots using re-dispatch. Ginger pre-allocates ROB and LSQ slots, and physical registers to the alternate path CD instructions of protected branches. However, it does not pre-allocate issue queue entries. Situations can arise in which logically older correct-path CD instructions are blocked from dispatching because all issue queue entries are held by logically younger instructions. In the rare situation when an un-executed wrong-path CD instruction with a large output dependence tree is removed from the window, deadlock can result. To avoid deadlock without resorting to squashing large portions of the window—and to avoid having to distinguish deadlock from more benign scenarios—Ginger can temporarily evict instructions from the issue queue by sending them to the re-dispatch queue with their re-dispatch bit pre-set. This simple option is not triggered often.

3.6. Partial-Bandwidth Rewriting and Re-Dispatch

To this point, tag rewriting was presented as operating at full rename/dispatch/issue bandwidth and selective re-dispatch at full dispatch bandwidth. In practice, Ginger can perform tag rewriting at less than full bandwidth and use the remaining bandwidth for conventional instruction processing. Similarly, re-dispatch can be performed at less than full dispatch bandwidth. Tag rewriting and re-dispatch can even be performed at different bandwidths. Partial-bandwidth rewriting and re-dispatch do not generally degrade performance because CI_{DD} instructions are not performance critical. In fact, because of this they may even improve performance. They also allow the re-dispatch queue and the ROB to be provisioned with fewer associative match ports.

Partial-bandwidth tag rewriting requires a few modifications to the map table and checkpoints. Under full-bandwidth re-writing, the active map table explicitly contains the "new" mappings and

does not need to be updated. If partial-bandwidth rewriting is used, the new mappings will be in one of the checkpoints and the active map-table may need to be updated as well. This configuration requires an additional signal per checkpoint bank to explicitly identify the new mapping checkpoint. It also requires copies of the tag-rewriting logic—the comparators and write enable signals—in the active map table. The active map table does not need additional read or write ports, however, because rewriting uses the ports that are not used by instructions renaming.

Partial-bandwidth tag rewriting allows the map table checkpoints to be provisioned with fewer read and write ports, and the ROB and re-dispatch queue to be provisioned with fewer pipelined associative match/replace ports. Because the re-dispatch queue match/replace ports are used both for tag rewriting and re-dispatch scheduling, providing fewer of these ports also reduces re-dispatch bandwidth. Ginger's default configuration uses 4-way rename, dispatch, and issue and 2-way tag-rewriting and re-dispatch.

3.7. Load Tag-Rewriting and Re-Dispatch

The re-dispatch queue re-dispatches all types of instructions and their dependents. In the presence of selective recovery, ensuring that instructions ultimately execute with the correct inputs requires a mechanism that repairs input data dependences and, in the process, identifies the first wave of instructions that must re-dispatch. For register dependences, tag rewriting is that mechanism. But loads may also have an implicit input in the form of an older in-flight store. When Ginger removes stores from the middle of the window and inserts others, affected loads must be identified and marked for re-dispatch.

Store queue index prediction (SQIP) [25] makes store-load forwarding look mechanically like speculative register communication. Store PCs are the analogs of logical register names. Store queue indices are the analogs of physical register names. The store map table, which maps store PCs to store queue indices, is the analog of the register map table. Therefore Ginger identifies loads that must re-dispatch using the analog of register tag rewriting. A bitvector tracks the store PCs (actually store map table indices) of wrong- and correct- path CD stores. Ginger walks this bitvector and uses the active store map table and the CI store map table checkpoint to perform store tag rewriting, replacing every instance of the old store queue index mapping with the new store queue index mapping and marking affected loads for re-dispatch.

SQIP is speculative and so store tag rewriting is as well. SQIP is verified and trained by SVW-filtered load re-execution prior to commit [3, 24]. In training SQIP's dependence predictor, SVW also effectively trains the store tag rewriting mechanism. SVW also verifies store tag rewriting by verifying the memory communication of the committed instruction stream. However, to support re-execution filtering for forwarding loads in the presence of out-of-order store removal, SVW needs a small extension. Under in-order SQIP, SVW filters re-executions for a forwarding load if the store queue index of the predicted forwarding store matches the store queue index of the last store to write to the load's address as recorded in the SVW bloom filter. With CI, a load can forward from a store, that store could be selectively-squashed and replaced with another store at the same store queue index and this new store may write an address that hashes to the same SVW bloom filter entry. Unmodified SVW would believe that the load forwarded from the correct store and forgo re-execution even though the load actually forwarded from the store that was squashed. To close this hole, SVW checks not only that the load forwarded from the right store queue index, but also that it executed after the forwarding store. This is done by extending both LSQ and SVW bloom filter entries with short physical time-stamps.

3.8. CI-Aware Branch Prediction

Many modern branch predictors rely on global branch history that is in-order, complete, and correct with respect to a given branch. Selective squash recovery intuitively interferes with this basic setup and, as a result, can potentially degrade the accuracy of global history based schemes. It would be counter-productive if Ginger induced more mis-predictions than it tolerated.

As an illustration, assume branch A from Figure 1 is initially predicted not-taken. When branch G is fetched, it is predicted with a history that includes a not-taken outcome for A and some outcome for branch B. If A is mis-predicted and selectively-recovered, then the G's history changes to have a taken outcome for A and no outcome at all for B. Should G be re-predicted with this new history? Which global history should G use to update the predictor: the one with which it was predicted or the correct one?

Walker repairs the global history and re-predicts branches as it re-renames all CI instructions. Skipper uses a history of committed branches for both prediction and update, a policy that does not require on-the-fly history updates or re-prediction, but which intuitively forfeits accuracy by ignoring correlations with nearby branches. SBR does not specify its policy. Ginger excludes the outcomes of protected branches from the global history. In the example, Ginger simply predicts branch G with branches A and B missing from its history. Ginger does this by checkpointing the history when branch A is initially fetched and using that history from A's convergence point (E) onwards. Excluding branches whose outcomes (or mere presence) in the global history may change avoids the need for on-the-fly global history updates. In turn, avoiding history updates avoids the need to re-predict branches. This scheme preserves prediction accuracy in the common case that a branch is not correlated with older protected branches—if G is not correlated with A or B. It is also equivalent to the conventional contiguous-history scheme when Ginger is not actively used.

Empirically, most branches prefer CI-aware history to contiguous history. However, branches with strong correlations to older protected branches do not. To avoid degrading prediction accuracy of these branches, Ginger backs up the CI-aware history predictor with a contiguous-history predictor and extends the chooser accordingly. The backup table can be small because the number of branches that prefer contiguous history is small and because the important correlations for these branches are nearby.

3.9. Suppressing False Mis-Predictions

Selective-squash recovery can induce false mis-predictions. A correctly predicted branch that is CI_{DD} on an older mis-predicted branch may initially execute with wrong inputs and trigger a spurious squash. When the older branch executes and initiates selective-recovery, the younger branch re-executes with correct inputs and triggers a second squash. In Figure 1, branch G is CI_{DD} on branch A and is vulnerable to false mis-predictions. Note, only CI schemes that use re-issue are vulnerable to false mis-predictions. Skipper does not suffer from false mis-predictions because it avoids re-issue by pre-synchronizing CI_{DD} instructions.

Table 1. Baseline processor configuration and CI support.

Baseline	Pipeline	21 stages: 1 predict, 3 fetch, 4 decode, 1 rename, 1 dispatch, 1 schedule, 3 register read/bypass, 1 execute, 3 register write/complete, 1 SVW, 3 re-execute/write-buffer, 1 commit, 1 free. 18 cycle minimum branch mis-prediction penalty. 4-way fetch, rename, issue (up to 4 simple integer, 2 complex integer/FP, 2 load, 1 store), and commit.
	Instruction window	512-entry re-order buffer and load-store queue. 576 physical registers. Indexed store-load forwarding uses a 4K-entry predictor [25] and 64-entry store map table. 64-entry issue queue. 16 register/store map table checkpoints.
	Memory hierarchy	64KB, 4-way associative, 3-cycle instruction and data caches. 4MB, 16-way associative, 20-cycle L2. 150-cycle main memory. 16B memory bus operates at one-quarter core frequency. 16 outstanding misses.
	Branch predictor	192Kb (24KB) 13-branch history 32K-entry gShare/bimodal hybrid. 4K-entry 4-way associative BTB, 32-entry RAS.
Ginger	Confidence & convergence	4K-entry path-insensitive confidence predictor [10]. Ginger setup attempted for branches that are less than 95% confident. Convergence PCs and distances computed statically and communicated via hints which are discarded at decode.
	Re-dispatch	512-entry, 8-segment re-dispatch queue. 2-way tag-rewriting and re-dispatch.

False mis-predictions primarily represent a lost opportunity. The cost of a selective-squash for an older mis-prediction plus a false selective- or full- squash for a younger correctly predicted branch should be less than the cost of a full-squash for the older branch. But false mis-predictions can also actively degrade performance, for instance when selective recovery from a mis-predicted branch with a resolution latency of 20 cycles triggers a false mis-prediction on a branch with a resolution latency of 40 cycles.

Identifying branches that are vulnerable to false mis-predictions and delaying their execution until it becomes completely non-speculative often has the negative side-effect of delaying the discovery of true mis-predictions. Ginger takes a more refined approach. It identifies branches that experience more false mis-predictions than true mis-predictions using a small, tagged table of up-down counters that is updated at commit. It then synchronizes these branches with only the immediately older Ginger-protected branch and its re-issue wave. Synchronization with the older branch is achieved by tagging the younger branch with the older branch's output physical register at rename—Ginger assigns physical registers to all instructions—and broadcasting that tag when the older branch executes. Synchronization with the re-issue wave is accomplished by holding the younger branch while tag rewriting or wakeup is taking place in an older re-dispatch queue segment.

4. EVALUATION

We use cycle-level simulation to measure Ginger's performance. The timing simulator uses the SimpleScalar Alpha AXP ISA and system call modules. Table 1 details its baseline configuration and the added Ginger support. The benchmarks are the SPECint2000, MediaBench and CommBench suites. The programs are compiled for Alpha EV6 using the Digital OSF compiler with -O4 optimizations. Benchmarks run to completion: SPECint programs on their training inputs with 2% periodic sampling; others on their large or only inputs with no sampling.

The three benchmarks suites have 46 programs between them. Although we report averages across entire suites, space limits preclude showing detailed data for all benchmarks. Figure 6 shows the branch mis-prediction rates for all benchmarks with bars broken down by convergence distance. From here on, we show detailed results for the five benchmarks from each suite with the largest percentage of mis-predicted branches that converge within 256 instructions. In Figure 6, the names of these programs are in bold.

4.1. Comparative Performance

Figure 7 shows speedups for Ginger (G) and re-implementations of Walker (W), SBR (S), and Skipper (K). The graph shows the selected benchmarks as well as the geometric mean (GM) over each suite. Under each benchmark name is its baseline IPC and the potential speedup obtained with perfect branch prediction. Under each bar is the percentage of mis-predicted branches to which CI is successfully applied. For example, *G:70* under the left-most bar means that Ginger successfully applies selective squash recovery to 70% of *crafty*'s mis-predicted branches.

Speedups for competing schemes do not always match up with originally reported numbers. This is largely due to baseline configuration differences. We have roughly reproduced the speedups reported in the original publications using the described configurations. The competing schemes are given the benefit of Ginger's support mechanisms: a large non-associative load-store queue, CI-aware branch prediction, and false mis-prediction suppression. Skipper is given Ginger's re-dispatch buffer which it uses to improve its management of issue queue entries.

For additional insight, Figure 8 shows a breakdown of the excess rename/dispatch bandwidth—defined as a slot used on anything other than an instruction that is eventually committed—for each CI scheme and for a conventional, non-CI processor. We use rename/dispatch rather than fetch bandwidth because CI operations like tag rewriting do not consume fetch bandwidth. Starting

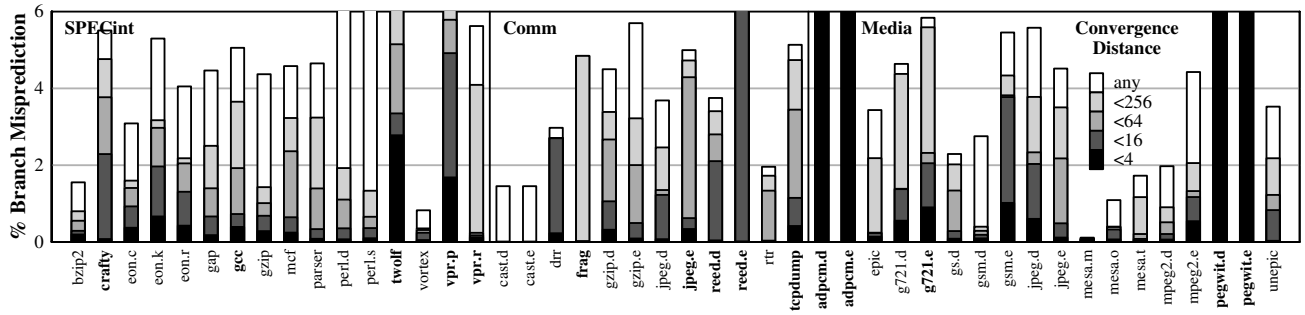


Figure 6. Branch mispredction rate by cumulative convergence distance.

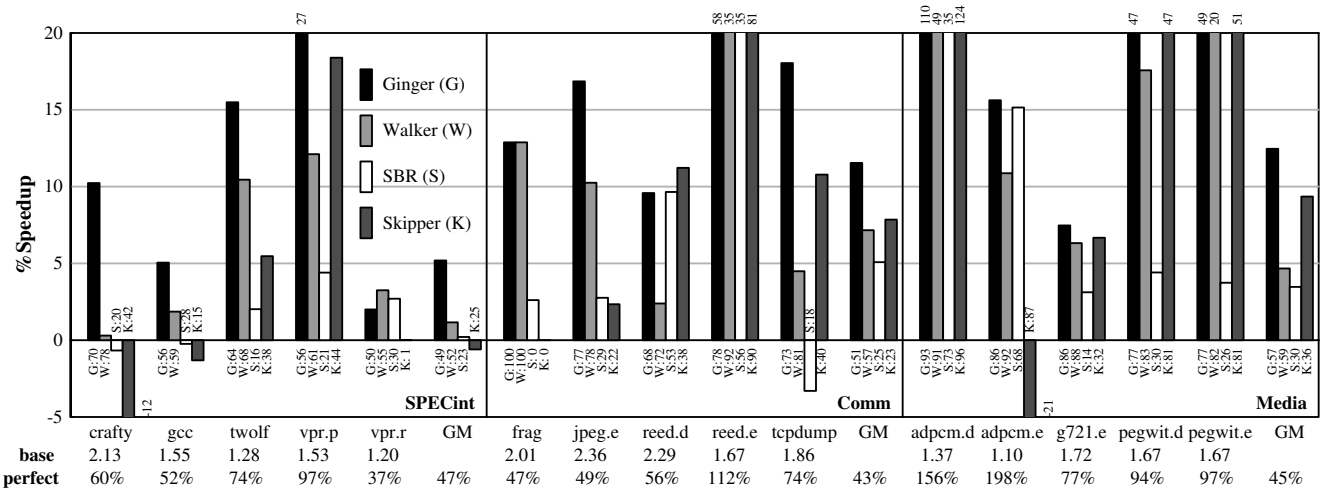


Figure 7. Relative performance comparison: Ginger, Walker, SBR, and Skipper.

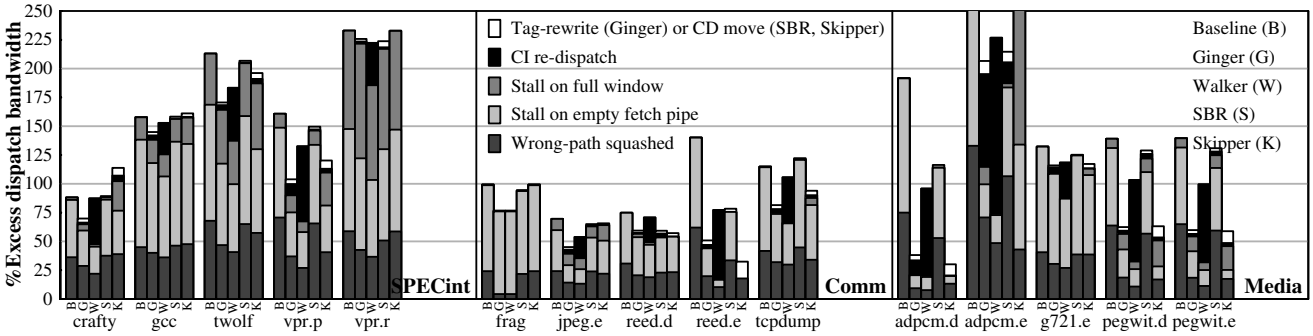


Figure 8. Excess dispatch bandwidth breakdown: Baseline, Ginger, Walker, SBR, and Skipper.

at the bottom of each bar in the stack, a conventional processor spends excess rename/dispatch bandwidth in three ways: (i) dispatching wrong-path instructions that are ultimately squashed, (ii) stalled due to an empty fetch pipe, and (iii) stalled due to back-pressure from a full re-order buffer or issue queue. With a 512-entry re-order buffer, 64-entry issue queue, and realistic branch prediction, the first two categories dominate most benchmarks. A CI processor can spend excess rename/dispatch bandwidth in two additional ways: (iv) re-dispatching CI instructions, and (v) changing tags (Ginger), re-dispatching CD instructions as moves (SBR), or dispatching post CD region moves (Skipper).

Ginger. Ginger achieves average performance gains of 5% on SPECint, 11% on Comm, and 12% on Media, with peaks of 27% (*vpr.p*), 58% (*reed.e*), and 110% (*adpcm.d*). Ginger successfully applies selective-squash recovery to 53% of all mis-predicted branches, reducing the number of instructions squashed due to mis-predictions (the bottom component of its excess dispatch bandwidth bar). Ginger also wastes fewer dispatch slots waiting for the front-end pipeline to fill after a mis-prediction. During selective squash recovery Ginger drains the front-end pipeline into the window rather than discarding those instructions.

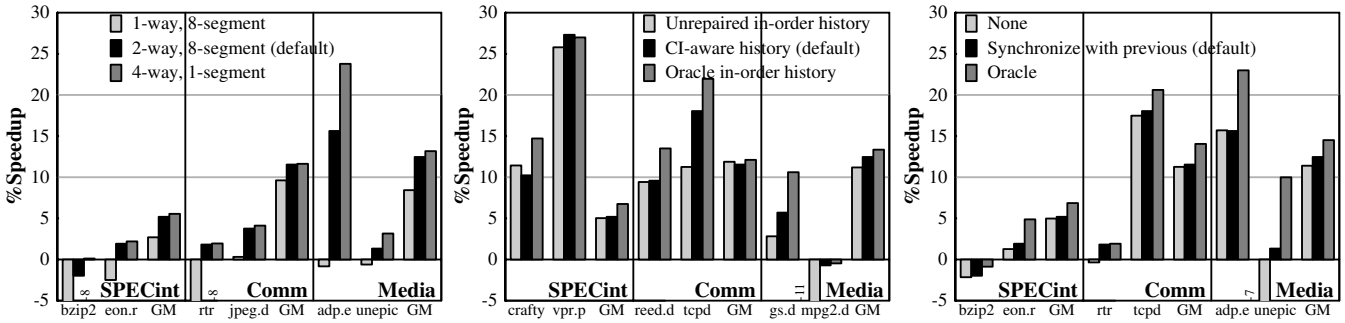
Ginger performs selective recovery with low rename/dispatch bandwidth overhead. This overhead is proportional to the number of CI_{DD} instructions (re-dispatch, black portion of each bar) plus the size of mis-predicted branch's actual—not conservatively speculative—register and store output set (tag rewriting, white portion of each bar). As Figure 8 shows, these two components *combined* are typically smaller than reductions in excess dispatch bandwidth spent on wrong-path instructions *alone*. Because it both improves window utilization and reserves window space for alternate-path CD instructions, Ginger can increase dispatch stalls due to a full ROB or issue queue (e.g., *vpr.r*).

Walker. Because it requires fewer checkpoints per selective squash maneuver, Walker applies selective squash recovery to slightly more mis-predicted branches than does Ginger—56% to 53% on average. However, Walker's speedups are lower—1%, 7%, and 4% on SPECint, Comm, and Media, with highs of 12% (*vpr.p*), 35% (*reed.e*), and 49% (*adpcm.d*)—primarily because it consumes more rename/dispatch bandwidth to repair dependences. This excess bandwidth is proportional to the number of CI instructions and is seen in Figure 8 as the large black portions of Walker's bars.

Walker was initially evaluated on a 16-way processor in which re-renaming all CI instructions is less prohibitive. It obtained relatively greater speedups in that configuration.

SBR. Of the three previously-proposed schemes, SBR has the lowest overall speedups: 0.2% on SPECint, 5% for Comm, and 4% on Media with a peak of 35% (*adpcm.d* and *reed.e)*. SBR's original evaluation used a 40-cycle minimum mis-prediction penalty and obtained higher speedups than shown here.

SBR applies selective recovery to fewer branches than Ginger—exact convergence occurs naturally in only 26% of mis-predicted branches and our SBR re-implementation does not induce it.



A) Re-dispatch queue implementation **B)** CI-aware branch prediction **C)** False mis-prediction suppression

Figure 9. Ginger mechanism sensistivity analysis.

Even in exact convergence cases, tag-rewriting consumes less bandwidth than re-dispatching wrong-path CD instructions as moves. Tag rewriting also adds no dataflow delay.

On *frag*, SBR produces an apparent anomaly—a 3% speedup with successful selective-squash recovery applied to 0% of mis-predicted branches! In actuality, 0.2% of mis-predicted branches are selectively recovered. Wrong-history branch predictor updates for these branches constructively interfere and slightly improve overall prediction accuracy. With a different predictor indexing function, the same level of activity produces only a 0.3% speedup.

Skipper. Skipper's speedups are –1% on SPECint, 8% on Comm, and 9% on Media. It can produce dramatic gains (124% on *adpcm.d* and 85% on *reed.e*) but also cause slowdowns (–12% on *crafty*). Skipper applies CI to only 29% of mis-predicted branches. Because skipping introduces delays even when the skipped branch is correctly predicted, our experiments indicate that Skipper provides the best overall results when applied to branches that are 80% confident or less. In contrast, Ginger targets branches that are less than 95% confident. The programs on which Skipper out-performs the reactive CI schemes all have dynamically frequent branches that mis-predict at a high rate. For instance, *reed.e*'s dominant low-confidence branch is the back-edge of a small loop and is frequently mis-predicted inside the skipped region. When these extremely low-confidence branches are clustered, Skipper can apply CI to more branches than Ginger (e.g., *adpcm.d* and *pegwit.d*). Again, because it uses fewer map table checkpoints.

Our Skipper re-implementation obtains lower speedups than originally reported because it uses an issue queue that is smaller than the ROB. However, it does benefit from the ability to use the re-dispatch queue to escape from local issue queue slot allocation inversions. Without a re-dispatch queue and with a smaller-than-ROB issue queue Skipper's average speedups would drop to –4%, 6%, and 5%, respectively.

Discussion: proactive and hybrid Ginger. Ginger can also operate in a proactive Skipper-like mode—skipping low-confidence branches, filling in the correct-path CD instructions after the branches resolves, and using tag-rewriting to update and re-dispatch CI_{DD} instructions. Proactive Ginger is somewhat less effective than Skipper. Because Ginger does not pre-synchronize CI_{DD} instructions, it must re-dispatch them whenever it performs proactive CI even if the branch is correctly predicted. Ginger can be augmented with simple pre-synchronization support—based on register ready bits rather than explicit move instructions—and this support lets proactive Ginger slightly out-perform Skipper. However, reactive Ginger both out-performs proactive Ginger and does not require the additional pre-synchronization support. A hybrid Ginger design which applies proactive CI to branches with very low confidence (less than 70%) and reactive CI to branches with moderately low confidence (70–95%) is also possible and does outperform both exclusive approaches. However, its relatively small gains over purely reactive Ginger—most mis-predicted branches are 85–95% confident and so hybrid Ginger rarely applies proactive CI—do not justify the additional support required for proactive CI and hybridization.

4.2. Sensitivity: Ginger CI Mechanisms

This section evaluates the impact of three of Ginger's CI mechanisms. Each graph in Figure 9 shows Ginger's relative performance on two benchmarks and the mean from each suite.

Re-dispatch queue implementation. Figure 9A shows three re-dispatch queue designs. Reducing re-dispatch bandwidth from 2-way to 1-way reduces speedups by an average of 2–4%, hurting programs like *adpcm.e* that re-dispatch many CI_{DD} instructions. Ginger's 2-way, 8-segment re-dispatch queue performs nearly as well as an idealized 4-way, 1-segment re-dispatch queue.

CI-aware branch prediction. Figure 9B measures the effect of Ginger's CI-aware branch prediction against a simple scheme that uses un-repaired in-order histories and an idealized scheme that predicts and updates each branch using an oracle in-order history—the history a conventional in-order fetch processor uses.

On average, Ginger's CI-aware history scheme performs only slightly better than the simple un-repaired in-order history scheme. On several benchmarks, it performs slightly worse. In *crafty*, for instance, the backup contiguous-history table is undersized. The benefit of the Ginger scheme is that it avoids the pathologies of un-repaired in-order histories, as is in *mpeg2.d*. CI-aware history can actually out-perform oracle in-order history. By excluding certain branches, it effectively extends the history length and allows more distant correlations to be exploited (e.g., *vpr.p*).

False mis-prediction suppression. Figure 9C shows the impact of Ginger's false mis-prediction suppression mechanism. Also shown are configurations with no false mis-prediction suppression and with oracle suppression. As in CI-aware branch prediction, Ginger's false mis-prediction suppression mechanism has a small overall positive impact. Its main contribution is that it eliminates pathological false mis-predictions that plague certain programs like *unepic*. As the oracle suppression results show, there is potential for further improvement in this area.

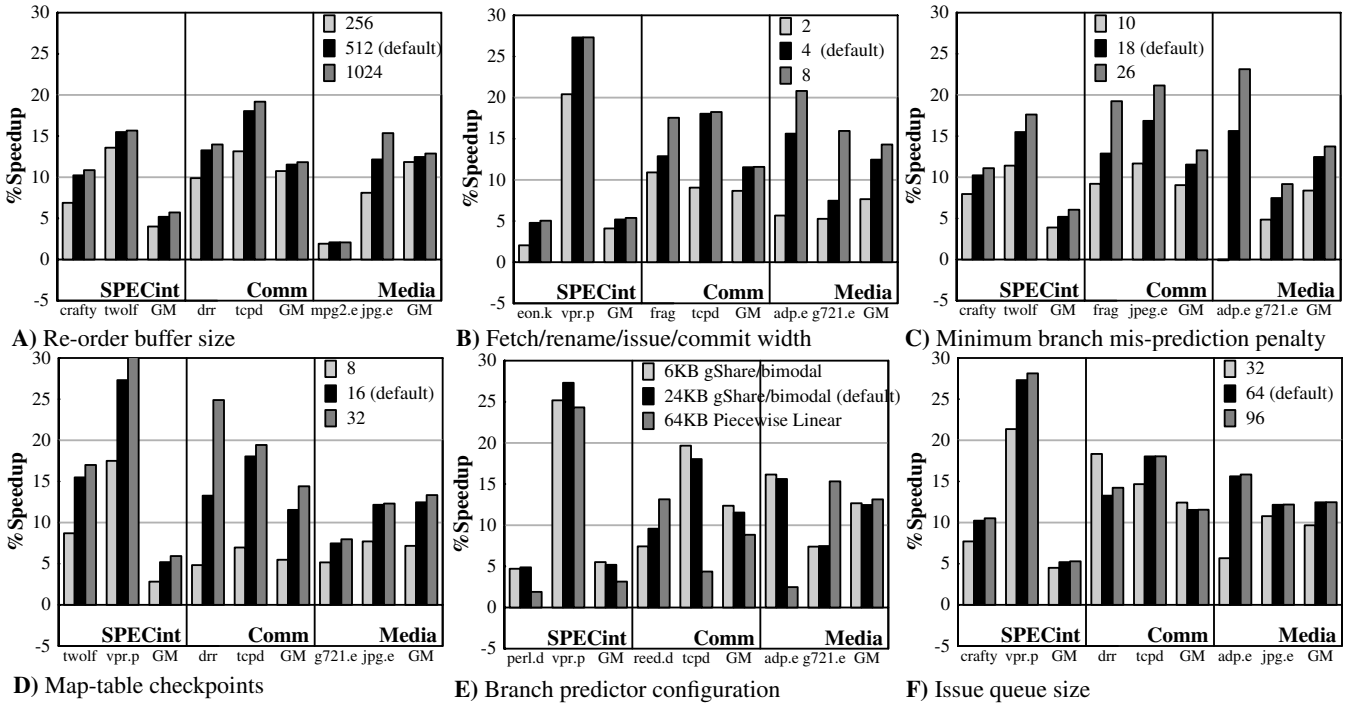


Figure 10. Baseline configuration sensitivity analysis.

4.3. Sensitivity: Baseline Configuration

Figure 10 shows the impact of six micro-architectural parameters on Ginger's relative performance.

Ginger speedups follow intuitive trends relative to re-order buffer size (Figure 10A), pipeline bandwidth (10B), and pipeline depth or minimum branch mis-prediction penalty (10C). Larger windows and wider and deeper pipelines increase the number of instruction slots lost to full-squash mis-prediction recovery. Because the size of CD regions is fixed, are relatively larger number of CI instructions are spared by selective squash recovery and the benefit of CI techniques in general—and of Ginger in particular—increases. A larger re-order buffer (10C) and a greater number of map-table checkpoints (10D) also allow Ginger to exploit CI more aggressively and effectively.

Branch predictor configuration. Figure 10E shows Ginger speedups on processors with three branch direction predictors: the default 24KB, 13-branch history gShare/bimodal hybrid, a smaller 6KB 11-branch history version of the same, and a 32KB 32-branch history piece-wise linear predictor [11]. Although in general the intuition that a more accurate branch predictor reduces Ginger's effectiveness is true, there are exceptions. A better predictor may improve the accuracy of branches that Ginger cannot protect more (relatively) than the accuracy of protected branches, amplifying Ginger's effects. It may also increase the confidence of certain branches above Ginger's threshold and allow Ginger to allocate checkpoints to branches that are more likely to mis-predict. Finally, the piece-wise linear design itself is more naturally resistant to out-of-order branch prediction effects and its accuracy degrades less when it is used in a CI environment.

Issue queue size. Figure 10F shows the impact of issue queue size on Ginger speedups. Ginger increases the effective size of the instruction window and generally benefits from a larger issue queue. However, Ginger also uses a hierarchical re-dispatch scheme which serves to temporarily enlarge the issue queue whenever a branch is selective-squash recovered. This effect can help programs that are bound by small issue queue sizes, and can even produce higher relative speedups in such configurations (e.g., *drr)*.

5. RELATED WORK

A 1992 study [16] showed that an architecture that serializes each instruction with all older mis-predicted branches (SP or SPeculative) has a fraction of the potential ILP as one that serializes each instruction only with older mis-predicted branches on which it is control dependent (SP-CD-MF or SPeculative Control-Dependent Multiple-Flow). SP corresponds to conventional in-order rename architectures. SP-CD-MF corresponds to superscalar CI architectures like Walker, SBR, Skipper, Ginger, and Trace Processors [23], and to speculative thread architectures that parallelize branch mis-predictions like Multiscalar [28]. CD-MF corresponds to dataflow architectures, which execute non-speculatively in control-dependence order [2].

Out-of-order renaming. Pre-synchronization based out-of-order renaming was first used in Multiscalar to support speculative threads [28]. More recent designs have used it to hide instruction cache miss latency [31], exploit proactive CI (Skipper) [4], parallelize fetch and rename for front-end bandwidth [20], exploit instruction slack (non-criticality) within a trace [19], and again to support speculative threads [18]. Pre-synchronization supports only out-of-order instruction insertion, which is sufficient to exploit speculative multithreading, which itself can be thought of as CI for high-confidence branches, e.g., loop branches. To support CI for low-confidence branches, reactive approaches are preferred. These require out-of-order instruction removal which in turn requires out-of-order renaming schemes based on physical re-renaming [22] or instruction annulment [8].

Stark's Ph.D. thesis describes a tag rewriting mechanism that uses additional match lines and age tags to rewrite issue queue entries and search to find map table checkpoints that need re-writing [30]. This mechanism supports out-of-order dispatch aimed at tolerating instruction cache misses.

Control-flow techniques that do not use out-of-order renaming. Squash reuse reduces branch mis-prediction penalty by avoiding re-issue—but not re-fetch or re-rename—of CI_{DI} instructions [5, 27]. Predication [21] and multi-path execution [15, 32] side-step mis-predictions but spend fetch, rename, and issue bandwidth on wrong-path instructions to do so. Dynamic predicate prediction [6], dynamic predication of un-predicated code [12, 14], and dynamic un-predication of predicated code [13] can adaptively reduce this overhead when branch prediction accuracy is high.

6. CONCLUSION

Reactive control independence (CI)—also known as selective-squash recovery—can mitigate the impact of branch mis-predictions and improve performance. The key implementation challenge of reactive CI is out-of-order renaming—repairing the input dependences of instructions that were spared by selective-squashing and re-issuing them with the correct inputs. Ginger is a general reactive CI microarchitecture that implements out-of-order renaming by re-writing the input dependence tags of instructions as they sit in the window. Ginger performs tag rewriting efficiently by temporarily halting the pipeline and combining the structure ports normally used for the disparate functions of rename, dispatch, and wakeup in a new way.

Cycle-level simulations show that Ginger out-performs previously proposed CI schemes, achieving average speedups of 5%, 12%, and 11%—on SPECint, MediaBench, and CommBench programs respectively—over an aggressive conventional baseline.

7. ACKNOWLEDGMENTS

We thank the reviewers for their suggestions and Eric Rotenberg for his help with the final version of this manuscript. This work was supported by NSF awards CCR-0238203 and CCF-0541292, and by a donation from Intel.

8. REFERENCES

[1] H. Akkary, R. Rajwar, and S. Srinivasan. "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors." In *Proc. 36th Int'l Symposium on Microarchitecture*, pages 423–434, Dec. 2003.

[2] Arvind and R. Nikhil. "Executing a Program on the MIT Tagged-Token Dataflow Architecture." *IEEE Transactions on Computers*, 39(3):300–318, Mar. 1990.

[3] H. Cain and M. Lipasti. "Memory Ordering: A Value Based Definition." In *Proc. 31st Int'l Symposium on Computer Architecture*, pages 90–101, Jun. 2004.

[4] C. Cher and T. Vijaykumar. "Skipper: A Microarchitecture for Exploiting Control-Flow Independence." In *Proc. 34th Int'l Symposium on Microarchitecture*, pages 4–15, Dec. 2001.

[5] Y. Chou, J. Fung, and J. Shen. "Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection." In *Proc. 1999 Int'l Conference on Supercomputing*, pages 109–118, Jun. 1999.

[6] W. Chuang and B. Calder. "Predicate Prediction for Efficient Out-of-Order Execution." In *Proc. 17th Int'l Conference on Supercomputing*, pages 183–192, Jun. 2003.

[7] J. Collins, D. Tullsen, and H. Wang. "Control Flow Optimization via Dynamic Reconvergence Prediction." In *Proc. 37th Int'l Symposium on Microarchitecture*, pages 129–140, Dec. 2004.

[8] A. Gandhi, H. Akkary, and S. Srinivasan. "Reducing Branch Misprediction Penalty via Selective Branch Recovery." In *Proc. 10th Int'l Symposium on High Performance Computer Architecture*, pages 254–265, Feb. 2004.

[9] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. "The Optimal Logic Depth per Pipeline Stage is 6-8 F04 Inverter Delays." In *Proc. 29th Int'l Symposium on Computer Architecture*, pages 14–24, May 2002.

[10] E. Jacobson, E. Rotenberg, and J. Smith. "Assigning Confidence to Conditional Branch Predictions." In *Proc. 29th Int'l Symposium on Microarchitecture*, pages 142–152, Dec. 1996.

[11] D. Jimenez. "Piecewise Linear Branch Prediction." In *Proc. 32nd Int'l Symposium on Computer Architecture*, pages 382–393, Jun. 2005.

[12] H. Kim, J. Joao, O. Mutlu, and Y. Patt. "Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-Flow Graphs Based on Frequently Executed Paths." In *Proc. 39th Int'l Symposium on Microarchitecture*, pages 53–64, Dec. 2006.

[13] H. Kim, O. Mutlu, J. Stark, and Y. Patt. "Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution." In *Proc. 38th Int'l Symposium on Microarchitecture*, pages 43–54, Nov. 2005.

[14] A. Klauser, T. Austin, D. Grunwald, and B. Calder. "Dynamic Hammock Predication for Non-Predicated Instruction Set Architectures." In *Proc. 18th Annual Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 278–285, Oct. 1998.

[15] A. Klauser, A. Paithankar, and D. Grunwald. "Selective Eager Execution on the PolyPath Architecture." In *Proc. 25th Int'l Symposium on Computer Architecture*, pages 250–259, Jun. 1998.

[16] M. Lam and R. Wilson. "Limits of Control Flow on Parallelism." In *Proc. 19th Int'l Symposium on Computer Architecture*, pages 47–57, May 1992.

[17] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. "A Large, Fast Instruction Window for Tolerating Cache Misses." In *Proc. 29th Int'l Symposium on Computer Architecture*, pages 59–70, May 2002.

[18] K. Malik, K. Woley, S. Stone, M. Agarwal, V. Dhar, and M. Frank. "Confidence Based Out-of-Order Renaming for Speculatively Multithreaded Processors." Technical Report UILU-ENG-06-2208, University of Illinois, Jun. 2006.

[19] G. Muthler, D. Crowe, S. Patel, and S. Lumetta. "Instruction Fetch Deferral Using Static Slack." In *Proc. 35th Int'l Symposium on Microarchitecture*, pages 51–61, Nov. 2002.

[20] P. Oberoi and G. Sohi. "Parallelism in the Front-End." In *Proc. 30th Int'l Symposium on Computer Architecture*, pages 230–240, Jun. 2003.

[21] D. Pnevmatikatos and G. Sohi. "Guarded Execution and Dynamic Branch Prediction in Dynamic Instruction-Level Parallel Processors." In *Proc. 21st Int'l Symposium on Computer Architecture*, pages 120–129, Apr. 1994.

[22] E. Rotenberg, Q. Jacobsen, and J. Smith. "A Study of Control Independence in Superscalar Processors." In *Proc. 5th Int'l Symposium on High Performance Computer Architecture*, pages 115–124, Jan. 1999.

[23] E. Rotenberg and J. Smith. "Control Independence in Trace Processors." In *Proc. 32nd Int'l Symposium on Microarchitecture*, pages 4–15, Nov. 1999.

[24] A. Roth. "Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization." In *Proc. 32nd Int'l Symposium on Computer Architecture*, pages 458–468, Jun. 2005.

[25] T. Sha, M. Martin, and A. Roth. "Scalable Store-Load Forwarding via Store Queue Index Prediction." In *Proc. 38th Int'l Symposium on Microarchitecture*, pages 159–170, Nov. 2005.

[26] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. "POWER5 System Microarchitecture." *IBM Journal of Research and Development*, 49(4/5):505–525, May 2005.

[27] A. Sodani and G. Sohi. "Dynamic Instruction Reuse." In *Proc. 24th Int'l Symposium on Computer Architecture*, pages 194–205, Jun 1997.

[28] G. Sohi, S. Breach, and T. Vijaykumar. "Multiscalar Processors." In *Proc. 22nd Int'l Symposium on Computer Architecture*, pages 414–425, Jun. 1995.

[29] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. "Continual Flow Pipelines." In *Proc. 11th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, Oct. 2004.

[30] J. Stark. *Out-of-Order Fetch, Decode, and Issue*. PhD thesis, University of Michigan, Nov. 1999.

[31] J. Stark, P. Racunas, and Y. Patt. "Reducing the Performance Impact of Instruction Cache Misses by Writing Instructions into the Reservation Stations Out-of-Order." In *Proc. 30th Int'l Symposium on Microarchitecture*, pages 34–43, Dec. 1997.

[32] A. Uht and V. Sindagi. "Disjoint Eager Execution: an Optimal Form of Speculative Execution." In *Proc. 28th Int'l Symposium on Microarchitecture*, pages 313–325, Nov. 1995.